detailed description of the applied reference in support of applicant's arguments.

Rather, the Examiner applies a piecemeal interpretation of the applied reference in order to create the fiction that Hoyer teaches the asserted claimed features.

For example, independent claim 1 specifies:

> first receiving, by the server and from a user browser according to hypertext transport protocol (HTTP), a web-based user request specifying execution of a management operation by at least one selected host computer specified in the web-based user request ...

The Examiner cannot find a consistent teaching in Hoyer that teaches this simple feature, and therefore misapplies Hoyer in an attempt to urge the this claimed feature is taught by Hoyer.

For example, the citation of the "user browser" 210 of Fig. 5 belies the fact that Hoyer teaches that the element 210 is **not** a browser that outputs an HTTP web-based user request specifying execution of a management operation as asserted by the Examiner, but a ***Client-Side Performance Monitor*** that is implemented as a Java Applet:

> As discussed in detail below, the client side component 210 is a platform independent program and is preferably a Java applet which receives requested performance measurements from the server side component 225 and stores the performance measurements in a data cache for display.
>
> (Col. 6, line 64 to col.7, line 2).
>
> The client side component 210 of the performance monitor (PM) 200 uses Java applets for displaying the web server performance data. The Java applets interact with the server side component 225 running on each of the web servers 220, 320 (also called a CM/CC/PM node).
>
> (Col. 8, lines 29-33).
>
> Functionality is provided by the client side component 210 using a Java applet that implements the graphical user interface or GUI 500. The GUI 500 is supported by a data manager 510 which is responsible for collecting and storing performance measurements. A data cache is filled with performance data by a server side program, pmServ 550.
>
> (Col. 10, lines 51-55).

BEST AVAILABLE COPY

In fact, Hoyer stresses that the Java applet (e.g., the data manager 510 in the client performance monitor 210 of Fig. 5) needs to maintain a persistent socket connection with the listening resource 560 in the server performance monitor 225:

> The client side component 210 is connected via a socket to the server side component 225 which includes a pmServ thread 550, a pmListen thread 560, and a PM data collector thread 570.
>
> (Col. 11, lines 51-54).
>
> When monitoring begins, and the client side component 210 registers interest in receiving performance data updates for specified clusters, a start of activity time is set at this moment. The performance data is collected from the web servers in the specified cluster(s). The client side component 210 requests data of the server-side program 225 at a set time interval. If the performance data is to be saved, the collected data is written to a file. The main thread of pmServ 550 monitors a pmListen thread 560 and responds to isalive events sent to it from an availability watch program. The main thread of pmServ 550 starts the pmListen thread 560.
>
> The pmListen thread 560 of pmServ 550 listens for requests from the PMDataManager 510. ... When data collection is started, the pmListen thread 560 creates a data collection thread for each cluster that is being monitored. When pmServ 550 receives a stop data collection request, pmServ 550 stops each data collection thread. If data playback is requested, the pmListen thread uses the current data recording configuration to determine where to get the recorded data. When data is requested by the PMDataManager 510, the pmListen thread 560 retrieves the current performance data from pmServ data structures (not shown; placed there by each of the data collection threads 570).
>
> (Col. 11, line 66 to col. 12, line 26).

Hence, the performance monitor 210 relies on a persistent socket connection to retrieve performance data from the performance server monitor 225.

The Examiner also prefers to cite irrelevant portions of Hoyer et al. that have no relevance whatsoever to the disclosed monitoring of performance data by the disclosed client performance monitor 210: col. 5, lines 41-46 and 53-54 identify web server operations in general, and provide no teaching relevant to the client performance monitor 210 or the server performance monitor 225. As described above, the cited col. 8, lines 25-50 actually discloses use of Java applets in the client 210 for communications via a persistent socket connection with the server side component 225.

The cited col. 13, lines 33-60 and col. 15, lines 63-67 provide no disclosure whatsoever that the requests or responses are HTTP-based requests and responses. Rather, the cited col. 13, lines 33-60 (and column 14) describe a request/response sequence between the ***Java-based*** data manager 510 in the client performance monitor 210 and "pmServ" servlet in 550 the server performance monitor 225 using the persistent socket connection. As illustrated on pages 2-3 of the attached Exhibit A ("Lesson 1: Socket Communications" from the website "java.sun.com", 6 pages), the listenSocket Method creates a persistent connection for receipt of messages from a given client, for example by "loop[ing] on the input stream to read data as it comes in from the client" using an executable "do-while" loop.

However, as illustrated on page 4 of the attached Exhibit B ("Socket Programming HOWTO", 4 pages) , "a protocol like HTTP uses a socket for only one transfer." Hence, one skilled in the art would recognize that use of a socket for persistent communications precludes the use of HTTP, since any HTTP response would cause an immediate closure of that socket (i.e., HTTP uses a socket for only one transfer).

Further, there is no disclosure or suggestion of the claimed server element outputting to a selected host computer a web request according to HTTP protocol and that specifies a ***management command*** for execution of the ***management operation by the management resource*** of the at least one selected host computer; or receiving a web response according to HTTP protocol and that specifies *information based on execution of the* ***management operation***. All management requests originating from the server 220 (that includes the server performance monitor 225) of Hoyer are performed using Simple Network Management Protocol, which is notoriously well known in the art to be an application layer protocol that is distinct from HTTP (see, e.g., pages 2 and 4 of the attached Exhibit C ("TCP/IP Reference Page" from protocols.com, pages 1, 2, and 4).

Hoyer uses an HTTP Get request only to measure response times by the web servers in serving up "home pages". As described in the subject specification at page 10, lines 19-20, the claimed ***management operations*** are distinguishable from "application operations" in which a host computer is configured for providing a prescribed web service, such as outputting a web page in response to an HTTP Get request. Regardless, there is no disclosure or suggestion whatsoever that any HTTP request output to measure response times includes the claimed management command for execution of the management operation by the management resource of the at least one selected host computer, as claimed. Rather, the targeted web server simply outputs the HTTP request for processing by the web server resource, and not the management resource (which is described in Hoyer as operating according to SNMP, not HTTP).

Hence, Hoyer neither discloses using HTTP for requesting ***management operations***, or for providing ***information based on execution of the management operations.***

Both Hoyer and Curley et al. describe that HTTP is to be used for measuring response times of the web server application (see para. 195 of Curley et al. "this calculation represents how quickly the web server can turn around a client request."). Hence, the hypothetical combination simply provides the embodiment of Hoyer, where an HTTP Post may be used instead of an HTTP Get request in order to measure web server response times.

Hence, the hypothetical combination provides no disclosure or suggestion of a server receiving an HTTP request specifying execution of a management operation, let alone outputting from the web server a web request specifying a management command for execution of the management operation ***by the management resource of the at least one selected host computer***, as claimed.

In fact, one having ordinary skill in the art would avoid using HTTP Posts, as claimed, because the instant destruction of a socket after a single transfer would render the Hoyer embodiment inoperable and change the principle operation of Hoyer by removing the persistent socket connection between the data manager 510 of the client performance monitor 210 and the server resources 550 and 560 of the server performance monitor 225 (see Fig. 5 of Hoyer). The Examiner is reminded that the proposed modification cannot change the principle operation of a reference or render it unsatisfactory for its intended purpose. "If the proposed modification or combination of the prior art would change the principle of operation of the prior art invention being modified, then the teachings of the references are not sufficient to render the claims prima facie obvious." MPEP § 2143.02, Rev. 2, May 2004 at p. 2100-132 (Citing In re Ratti, 270 F.2d 810, 123 USPQ 349 (CCPA 1959). "If the proposed modification would render the prior art invention being modified unsatisfactory for its intended purpose, then there is no suggestion or motivation to make the proposed modification." Id. (Citing In re Gordon, 733 F.2d 900, 221 USPQ 1125 (Fed. Cir. 1984)). Cf. MPEP §2145.III at page 2100-160 (Rev. 2, May 2004) ("the claimed combination cannot change the principle of operation of the primary reference or render the reference inoperable for its intended purpose.").

Moreover, the hypothetical combination still would neither disclose nor suggest that the claimed server would output ***an HTTP Post*** to a selected host computer specifying a ***management command***, or receive an ***HTTP Post*** from the selected host computer specifying information based on ***execution of the management operation*** according to the management command, as claimed. Rather, the hypothetical combination at most would apply an HTTP post between the user browser and the server, but not between the server and other host computers for management operations, as claimed.

For these and other reasons, the §103 rejection should be withdrawn.

It is believed that the remaining dependent claims are allowable in view of the foregoing.

In view of the above, it is believed this application is and condition for allowance, and such a Notice is respectfully solicited.

To the extent necessary, Applicant petitions for an extension of time under 37 C.F.R. 1.136. Please charge any shortage in fees due in connection with the filing of this paper, including any missing or insufficient fees under 37 C.F.R. 1.17(a), to Deposit Account No. 50-1130, under Order No. 95-463, and please credit any excess fees to such deposit account.

Respectfully submitted,

Leon R. Turkevich
Registration No. 34,035

Customer No.23164
Date: January 3, 2006

http://java.sun.com/developer/onlineTraining/Programming/BasicJava2/socket.html Jan 02, 2006

Tutorials & Code Camps

Lesson 1: Socket Communications

Lesson 1: Socket Communications

[<<BACK] [CONTENTS] [NEXT>>]

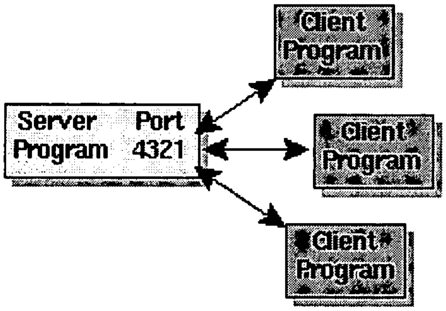
Java Programming Language Basics, Part 1, finished with a simple network communications example using the Remote Method Invocation (RMI) application programming interface (API). The RMI example allows multiple client programs to communicate with the same server program without any explicit code to do this because the RMI API is built on sockets and threads.

This lesson presents a simple sockets-based program to introduce the concepts of sockets and multi-threaded programming. A multi-threaded program performs multiple tasks at one time such as fielding simultaneous requests from many client programs.

> **Note:** See Creating a Threaded Slide Show Applet for another example of how multiple threads can be used in a program.

What are Sockets and Threads?

A socket is a software endpoint that establishes bidirectional communication between a server program and one or more client programs. The socket associates the server program with a specific hardware port on the machine where it runs so any client program anywhere in the network with a socket associated with that same port can communicate with the server program.



A server program typically provides resources to a network of client programs. Client programs send requests to the server program, and the server program responds to the request.

One way to handle requests from more than one client is to make the server program multi-threaded. A multi-threaded server creates a thread for each communication it accepts from a client. A thread is a sequence of instructions that run independently of the program and of any other threads.

Using threads, a multi-threaded server program can accept a connection from a client, start a thread for that communication, and continue listening for requests from other clients.

About the Examples

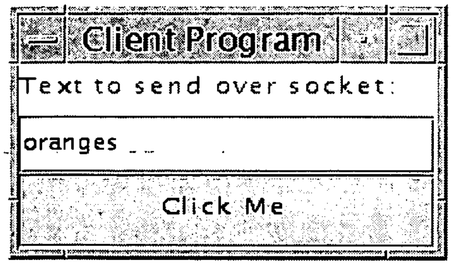
The examples for this lesson consist of two versions of the client and server program pair adapted from the FileIO.java application presented in Part 1, Lesson 6: File Access and Permissions.

Example 1 sets up a client and server communication between one server program and one client program. The server program is not multi-threaded and cannot handle requests from more than one client.

Example 2 converts the server program to a multi-threaded version so it can handle requests from more than one client.

Example 1: Client-Side Behavior

The client program presents a simple user interface and prompts for text input. When you click the Click Me button, the text is sent to the server program. The client program expects an echo from the server and prints the echo it receives on its standard output.

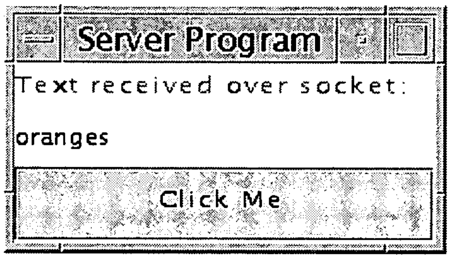


Text to send over socket:

oranges

Example 1: Server-Side Behavior

The server program presents a simple user interface, and when you click the Click Me button, the text received from the client is displayed. The server echoes the text it receives whether or not you click the Click Me button.



Text received over socket:

oranges

Example 1: Compile and Run

To run the example programs, start the server program first. If you do not, the client program cannot establish the socket connection. Here are the compiler and interpreter commands to compile and run the example.

```
javac SocketServer.java
javac SocketClient.java

java SocketServer
java SocketClient
```

Example 1: Server-Side Program

The server program establishes a socket connection on Port 4321 in its listenSocket method. It reads data sent to it and sends that same data back to the server in its actionPerformed method.

listenSocket Method

The listenSocket method creates a ServerSocket object with the port number on which the server program is going to listen for client communications. The port number must be an available port, which means the number cannot be reserved or already in use. For example, Unix systems reserve ports 1 through 1023 for administrative functions leaving port numbers greater than 1024 available for use.

```
public void listenSocket(){
  try{
    server = new ServerSocket(4321);
  } catch (IOException e) {
    System.out.println("Could not listen on port 4321");
    System.exit(-1);
  }
```

Next, the listenSocket method creates a Socket connection for the requesting client. This code executes when a client starts up and requests the connection on the host and port where this server program is running. When the connection is successfully established, the server.accept method returns a new Socket object.

```
  try{
    client = server.accept();
  } catch (IOException e) {
    System.out.println("Accept failed: 4321");
    System.exit(-1);
  }
```

Then, the listenSocket method creates a BufferedReader object to read the data sent over the socket connection from the client program. It also creates a PrintWriter object to send the data received from the client back to the server.

```
  try{
   in = new BufferedReader(new InputStreamReader(
                   client.getInputStream()));
   out = new PrintWriter(client.getOutputStream(),
                 true);
  } catch (IOException e) {
    System.out.println("Read failed");
    System.exit(-1);
  }
}
```

Lastly, the listenSocket method loops on the input stream to read data as it comes in from the client and writes to the output stream to send the data back.

```
    while(true){
      try{
        line = in.readLine();
//Send data back to client
        out.println(line);
      } catch (IOException e) {
        System.out.println("Read failed");
        System.exit(-1);
      }
    }
```

actionPerformed Method

The actionPerformed method is called by the Java platform for action events such as button clicks. This actionPerformed method uses the text stored in the line object to initialize the textArea object so the retrieved text can be displayed to the end user.

```
public void actionPerformed(ActionEvent event) {
   Object source = event.getSource();

   if(source == button){
      textArea.setText(line);
   }
}
```

Example 1: Client-Side Program

The client program establishes a connection to the server program on a particular host and port number in its listenSocket method, and sends the data entered by the end user to the server program in its actionPerformed method. The actionPerformed method also receives the data back from the server and prints it to the command line.

listenSocket Method

The listenSocket method first creates a Socket object with the computer name (kq6py) and port number (4321) where the server program is listening for client connection requests. Next, it creates a PrintWriter object to send data over the socket connection to the server program. It also creates a BufferedReader object to read the text sent by the server back to the client.

```
public void listenSocket(){
//Create socket connection
   try{
     socket = new Socket("kq6py", 4321);
     out = new PrintWriter(socket.getOutputStream(),
```

Exhibit A to Response Filed January 3, 2006, Appln. No. 09/817,055 (Page 3 of 6)

```
                true);
        in = new BufferedReader(new InputStreamReader(
                socket.getInputStream()));
      } catch (UnknownHostException e) {
        System.out.println("Unknown host: kq6py");
        System.exit(1);
      } catch  (IOException e) {
        System.out.println("No I/O");
        System.exit(1);
      }
   }
```

actionPerformed Method

The actionPerformed method is called by the Java platform for action events such as button clicks. This actionPerformed method code gets the text in the Textfield object and passes it to the PrintWriter object, which then sends it over the socket connection to the server program.

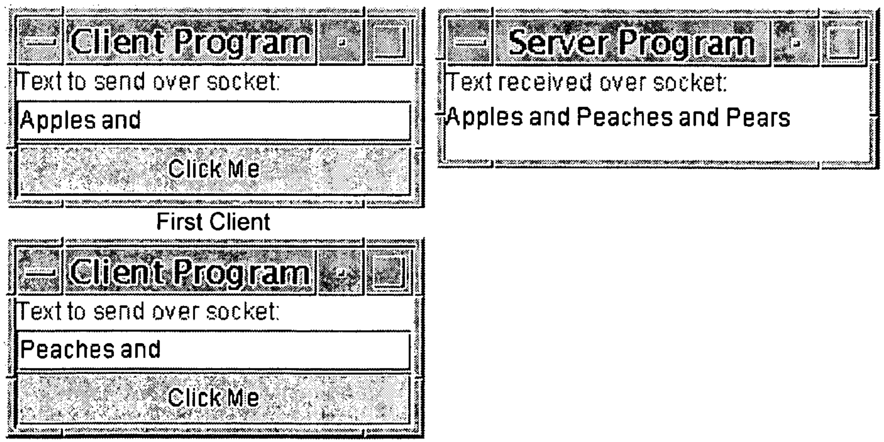
The actionPerformed method then makes the Textfield object blank so it is ready for more end user input. Lastly, it receives the text sent back to it by the server and prints the text out.

```
   public void actionPerformed(ActionEvent event){
     Object source = event.getSource();

     if(source == button){
   //Send data over socket
        String text = textField.getText();
        out.println(text);
        textField.setText(new String(""));
        out.println(text);
     }
   //Receive text from server
     try{
       String line = in.readLine();
       System.out.println("Text received: " + line);
     } catch (IOException e){
       System.out.println("Read failed");
       System.exit(1);
     }
   }
```

Example 2: Multithreaded Server Example

The example in its current state works between the server program and one client program only. To allow multiple client connections, the server program has to be converted to a multithreaded server program.



Text to send over socket:

Apples and

Text received over socket:

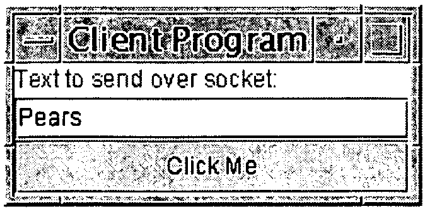
Apples and Peaches and Pears

First Client

Text to send over socket:

Peaches and

Second Client



Text to send over socket:
Pears

Third Client

The multithreaded server program creates a new thread for every client request. This way each client has its own connection to the server for passing data back and forth. When running multiple threads, you have to be sure that one thread cannot interfere with the data in another thread.

In this example the listenSocket method loops on the server.accept call waiting for client connections and creates an instance of the ClientWorker class for each client connection it accepts. The textArea component that displays the text received from the client connection is passed to the ClientWorker instance with the accepted client connection.

```
 public void listenSocket(){
  try{
    server = new ServerSocket(4444);
  } catch (IOException e) {
    System.out.println("Could not listen on port 4444");
    System.exit(-1);
  }
  while(true){
    ClientWorker w;
    try{
//server.accept returns a client connection
      w = new ClientWorker(server.accept(), textArea);
      Thread t = new Thread(w);
      t.start();
    } catch (IOException e) {
      System.out.println("Accept failed: 4444");
      System.exit(-1);
    }
  }
 }
```

The important changes in this version of the server program over the non-threaded server program are the line and client variables are no longer instance variables of the server class, but are handled inside the ClientWorker class.

The ClientWorker class implements the Runnable interface, which has one method, run. The run method executes independently in each thread. If three clients request connections, three ClientWorker instances are created, a thread is started for each ClientWorker instance, and the run method executes for each thread.

In this example, the run method creates the input buffer and output writer, loops on the input stream waiting for input from the client, sends the data it receives back to the client, and sets the text in the text area.

```
 class ClientWorker implements Runnable {
  private Socket client;
  private JTextArea textArea;

 //Constructor
  ClientWorker(Socket client, JTextArea textArea) {
    this.client = client;
    this.textArea = textArea;
  }

  public void run(){
    String line;
    BufferedReader in = null;
    PrintWriter out = null;
    try{
      in = new BufferedReader(new
        InputStreamReader(client.getInputStream()));
      out = new
        PrintWriter(client.getOutputStream(), true);
      } catch (IOException e) {
        System.out.println("in or out failed");
        System.exit(-1);
      }

      while(true){
        try{
          line = in.readLine();
  //Send data back to client
          out.println(line);
  //Append data to text area
          textArea.append(line);
        }catch (IOException e) {
          System.out.println("Read failed");
          System.exit(-1);
        }
      }
    }
  }
```

The JTextArea.append method is thread safe, which means its implementation includes code that allows one thread to finish its append operation before another thread can start an append operation. This prevents one thread from overwriting all or part of a string of appended text and corrupting the output. If the JTextArea.append method were not thread safe, you would need to wrap the call to textArea.append(line) in a synchronized method and replace the run method call to textArea.append(line) with a call to appendText(line):

```
  public synchronized void appendText(line){
    textArea.append(line);
  }
```

The synchronized keyword means this thread has a lock on the textArea and no other thread can change the textArea until this thread finishes its append operation.

The finalize() method is called by the Java virtual machine (JVM)* before the program exits to give the program a chance to clean up and release resources. Multi-threaded programs should close all Files and Sockets they use before exiting so they do not face resource starvation. The call to server.close() in the finalize() method closes the Socket connection used by each thread in this program.

```
    protected void finalize(){
  //Objects created in run method are finalized when
  //program terminates and thread exits
       try{
          server.close();
      } catch (IOException e) {
          System.out.println("Could not close socket");
          System.exit(-1);
      }
    }
```

More Information

You can find more information on sockets in the All About Sockets section in The Java Tutorial.

[TOP]

copyright © Sun Microsystems, Inc

Socket Programming HOWTO

Gordon McMillan

gmcm@hypernet.com

Abstract:

Sockets are used nearly everywhere, but are one of the most severely misunderstood technologies around. This is a 10,000 foot overview of sockets. It's not really a tutorial - you'll still have work to do in getting things operational. It doesn't cover the fine points (and there are a lot of them), but I hope it will give you enough background to begin using them decently.

This document is available from the Python HOWTO page at http://www.python.org/doc/howto.

Contents

1 Sockets

Sockets are used nearly everywhere, but are one of the most severely misunderstood technologies around. This is a 10,000 foot overview of sockets. It's not really a tutorial - you'll still have work to do in getting things working. It doesn't cover the fine points (and there are a lot of them), but I hope it will give you enough background to begin using them decently.

I'm only going to talk about INET sockets, but they account for at least 99% of the sockets in use. And I'll only talk about STREAM sockets - unless you really know what you're doing (in which case this

HOWTO isn't for you!), you'll get better behavior and performance from a STREAM socket than anything else. I will try to clear up the mystery of what a socket is, as well as some hints on how to work with blocking and non-blocking sockets. But I'll start by talking about blocking sockets. You'll need to know how they work before dealing with non-blocking sockets.

Part of the trouble with understanding these things is that "socket" can mean a number of subtly different things, depending on context. So first, let's make a distinction between a "client" socket - an endpoint of a conversation, and a "server" socket, which is more like a switchboard operator. The client application (your browser, for example) uses "client" sockets exclusively; the web server it's talking to uses both "server" sockets and "client" sockets.

1.1 History

Of the various forms of IPC (*Inter Process Communication*), sockets are by far the most popular. On any given platform, there are likely to be other forms of IPC that are faster, but for cross-platform communication, sockets are about the only game in town.

They were invented in Berkeley as part of the BSD flavor of Unix. They spread like wildfire with the Internet. With good reason -- the combination of sockets with INET makes talking to arbitrary machines around the world unbelievably easy (at least compared to other schemes).

2 Creating a Socket

Roughly speaking, when you clicked on the link that brought you to this page, your browser did something like the following:

```
#create an INET, STREAMing socket
s = socket.socket(
    socket.AF_INET, socket.SOCK_STREAM)
#now connect to the web server on port 80
# - the normal http port
s.connect(("www.mcmillan-inc.com", 80))
```

When the `connect` completes, the socket `s` can now be used to send in a request for the text of this page. The same socket will read the reply, and then be destroyed. That's right - destroyed. Client sockets are normally only used for one exchange (or a small set of sequential exchanges).

What happens in the web server is a bit more complex. First, the web server creates a "server socket".

```
#create an INET, STREAMing socket
serversocket = socket.socket(
    socket.AF_INET, socket.SOCK_STREAM)
#bind the socket to a public host,
# and a well-known port
serversocket.bind((socket.gethostname(), 80))
#become a server socket
serversocket.listen(5)
```

A couple things to notice: we used `socket.gethostname()` so that the socket would be visible to the outside world. If we had used `s.bind(('', 80))` or `s.bind(('localhost', 80))` or `s.bind(('127.0.0.1', 80))` we would still have a "server" socket, but one that was only visible within the same machine.

A second thing to note: low number ports are usually reserved for "well known" services (HTTP, SNMP etc). If you're playing around, use a nice high number (4 digits).

Finally, the argument to `listen` tells the socket library that we want it to queue up as many as 5 connect requests (the normal max) before refusing outside connections. If the rest of the code is written properly, that should be plenty.

OK, now we have a "server" socket, listening on port 80. Now we enter the mainloop of the web server:

```
while 1:
    #accept connections from outside
    (clientsocket, address) = serversocket.accept()
    #now do something with the clientsocket
    #in this case, we'll pretend this is a threaded server
    ct = client_thread(clientsocket)
    ct.run()
```

There's actually 3 general ways in which this loop could work - dispatching a thread to handle `clientsocket`, create a new process to handle `clientsocket`, or restructure this app to use non-blocking sockets, and mulitplex between our "server" socket and any active `clientsockets` using `select`. More about that later. The important thing to understand now is this: this is *all* a "server" socket does. It doesn't send any data. It doesn't receive any data. It just produces "client" sockets. Each `clientsocket` is created in response to some *other* "client" socket doing a `connect()` to the host and port we're bound to. As soon as we've created that `clientsocket`, we go back to listening for more connections. The two "clients" are free to chat it up - they are using some dynamically allocated port which will be recycled when the conversation ends.

2.1 IPC

If you need fast IPC between two processes on one machine, you should look into whatever form of shared memory the platform offers. A simple protocol based around shared memory and locks or semaphores is by far the fastest technique.

If you do decide to use sockets, bind the "server" socket to `'localhost'`. On most platforms, this will take a shortcut around a couple of layers of network code and be quite a bit faster.

3 Using a Socket

The first thing to note, is that the web browser's "client" socket and the web server's "client" socket are identical beasts. That is, this is a "peer to peer" conversation. Or to put it another way, *as the designer, you will have to decide what the rules of etiquette are for a conversation.* Normally, the `connecting` socket starts the conversation, by sending in a request, or perhaps a signon. But that's a design decision - it's not a rule of sockets.

Now there are two sets of verbs to use for communication. You can use `send` and `recv`, or you can transform your client socket into a file-like beast and use `read` and `write`. The latter is the way Java presents their sockets. I'm not going to talk about it here, except to warn you that you need to use `flush` on sockets. These are buffered "files", and a common mistake is to `write` something, and then `read` for a reply. Without a `flush` in there, you may wait forever for the reply, because the request may still be in your output buffer.

Now we come the major stumbling block of sockets - `send` and `recv` operate on the network buffers. They do not necessarily handle all the bytes you hand them (or expect from them), because their major focus is handling the network buffers. In general, they return when the associated network buffers have been filled (`send`) or emptied (`recv`). They then tell you how many bytes they handled. It is *your* responsibility to call them again until your message has been completely dealt with.

When a `recv` returns 0 bytes, it means the other side has closed (or is in the process of closing) the connection. You will not receive any more data on this connection. Ever. You may be able to send data successfully; I'll talk about that some on the next page.

A protocol like HTTP uses a socket for only one transfer. The client sends a request, the reads a reply. That's it. The socket is discarded. This means that a client can detect the end of the reply by receiving 0 bytes.

But if you plan to reuse your socket for further transfers, you need to realize that *there is no "EOT" (End of Transfer) on a socket.* I repeat: if a socket `send` or `recv` returns after handling 0 bytes, the connection has been broken. If the connection has *not* been broken, you may wait on a `recv` forever, because the socket will *not* tell you that there's nothing more to read (for now). Now if you think about that a bit, you'll come to realize a fundamental truth of sockets: *messages must either be fixed length* (yuck), *or be delimited* (shrug), *or indicate how long they are* (much better), *or end by shutting down the connection.* The choice is entirely yours, (but some ways are righter than others).

Assuming you don't want to end the connection, the simplest solution is a fixed length message:

```
class mysocket:
    '''demonstration class only
      - coded for clarity, not efficiency'''
    def __init__(self, sock=None):
        if sock is None:
            self.sock = socket.socket(
                socket.AF_INET, socket.SOCK_STREAM)
        else:
            self.sock = sock
    def connect(host, port):
        self.sock.connect((host, port))
    def mysend(msg):
        totalsent = 0
        while totalsent < MSGLEN:
            sent = self.sock.send(msg[totalsent:])
            if sent == 0:
                raise RuntimeError, \\
                    "socket connection broken"
            totalsent = totalsent + sent
    def myreceive():
        msg = ''
        while len(msg) < MSGLEN:
            chunk = self.sock.recv(MSGLEN-len(msg))
            if chunk == '':
                raise RuntimeError, \\
```

We have the solution for you!!!

RADCOM Academy

PROTOCOLS.COM

Search

Search
Protocol Directory
Hot Links
Acronyms
Tech Papers

TCP / IP Reference Page

Protocols according to layers

Data Link Layer
Network Layer
Transport Layer
Session Layer
Application Layer
Routing
Tunneling
Security

Click here to view protocols route map

Shortcuts to popular Protocols

IP	Internet Protocol version 4
IPv6	Internet Protocol version 6
TCP	Transmission Control Protocol
UDP	User Datagram Protocol

Cellular VoIP Protocol Analyzers Testing Solutions

Want to advertise on this site?

The Defense Advance Research Projects Agency (DARPA) originally developed Transmission Control Protocol/Internet Protocol (TCP/IP) to interconnect various defense department computer networks. The Internet, an international Wide Area Network, uses TCP/IP to connect government and educational institutions across the world. TCP/IP is also in widespread use on commercial and private networks. The TCP/IP suite includes the following protocols

Data Link Layer

ARP/RARP	Address Resolution Protocol/Reverse Address
DCAP	Data Link Switching Client Access Protocol

Network Layer

DHCP	Dynamic Host Configuration Protocol
DVMRP	Distance Vector Multicast Routing Protocol
ICMP/ICMPv6	Internet Control Message Protocol
IGMP	Internet Group Management Protocol
IP	Internet Protocol version 4
IPv6	Internet Protocol version 6
MARS	Multicast Address Resolution Server
PIM	Protocol Independent Multicast-Sparse Mode (PIM-SM)
RIP2	Routing Information Protocol
RIPng for IPv6	Routing Information Protocol for IPv6
RSVP	Resource ReSerVation setup Protocol

VRRP	Virtual Router Redundancy Protocol

Transport Layer

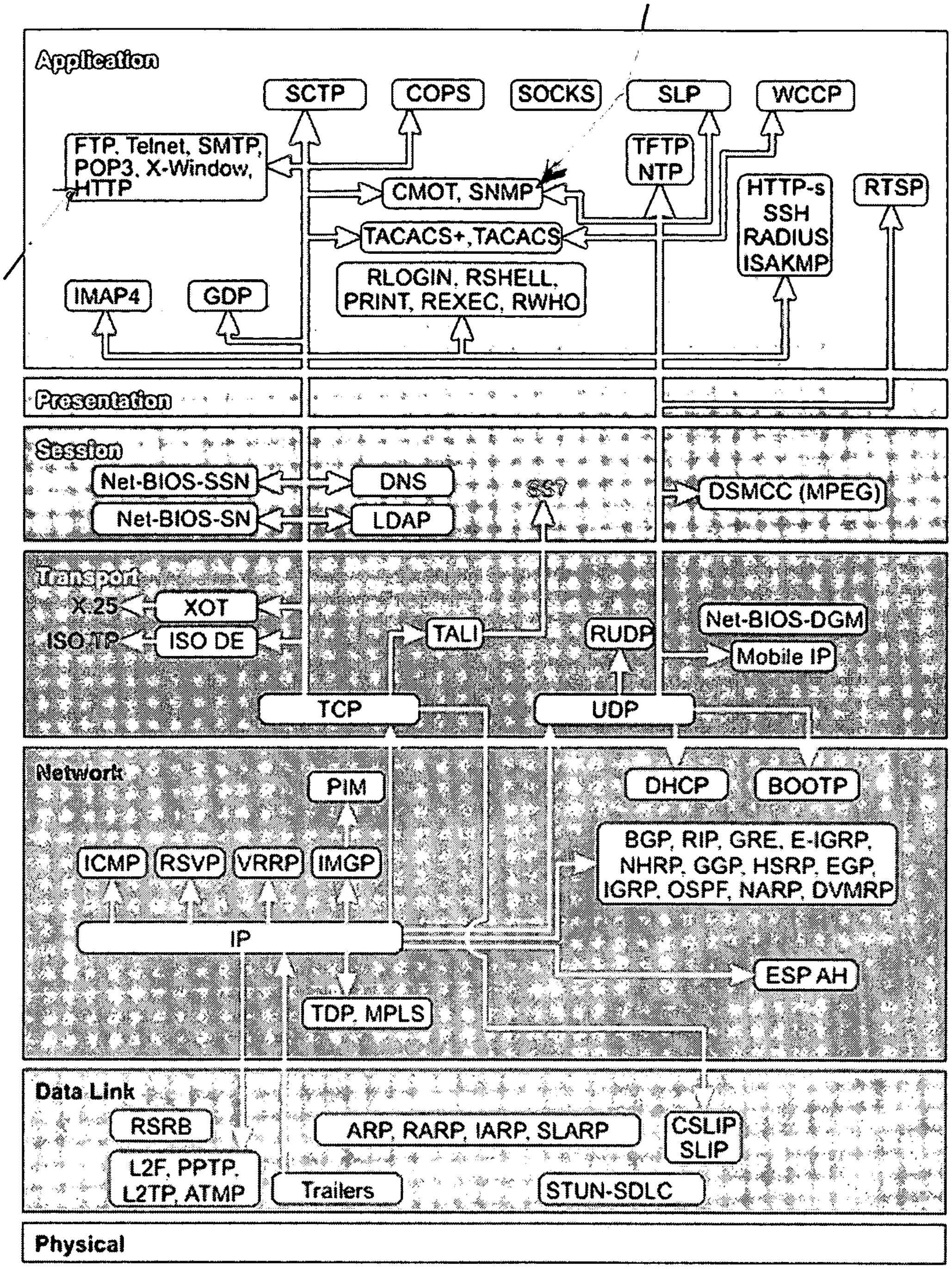
ISTP	
Mobile IP	Mobile IP Protocol
RUDP	Reliable UDP
TALI	Transport Adapter Layer Interface
TCP	Transmission Control Protocol
UDP	User Datagram Protocol
Van Jacobson	compressed TCP
XOT	X.25 over TCP

Session Layer

BGMP	Border Gateway Multicast Protocol
Diameter	
DIS	Distributed Interactive Simulation
DNS	Domain Name Service
ISAKMP/IKE	Internet Security Association and Key Management Protocol and Internet Key Exchange Protocol
iSCSI	Small Computer Systems Interface
LDAP	Lightweight Directory Access Protocol
MZAP	Multicast-Scope Zone Announcement Protocol
NetBIOS/IP	NetBIOS/IP for TCP/IP Environment

Application Layer

COPS	Common Open Policy Service
FANP	Flow Attribute Notification Protocol
Finger	User Information Protocol
FTP	File Transfer Protocol
HTTP	Hypertext Transfer Protocol ←
IMAP4	Internet Message Access Protocol rev 4
IMPPpre/IMPPmes	Instant Messaging and Presence Protocols
IPDC	IP Device Control
IRC	Internet Relay Chat Protocol
ISAKMP	Internet Message Access Protocol version 4rev1
ISP	
NTP	Network Time Protocol
POP3	Post Office Protocol version 3
Radius	Remote Authentication Dial In User Service
RLOGIN	Remote Login
RTSP	Real-time Streaming Protocol
SCTP	Stream Control Transmision Protocol
S-HTTP	Secure Hypertext Transfer Protocol
SLP	Service Location Protocol
SMTP	Simple Mail Transfer Protocol
SNMP	Simple Network Management Protocol ←
SOCKS	Socket Secure (Server)
TACACS+	Terminal Access Controller Access Control System
TELNET	TCP/IP Terminal Emulation Protocol
TFTP	Trivial File Transfer Protocol



1 2 3 4 5 6 7 8 9 ▷

TCP/IP Family Protocol Information

AH | ATMP | ARP/RARP | BGMP | BGP-4 | COPS | DCAP | DHCP | Diameter | DIS | DNS | DVMRP | EGP | EIGRP | ESP | FANP | Finger | FTP | HSRP | HTTP | ICMP/ICMPv6 | IGMP | IGRP | IMAP4 | IMPPpre/IMPPmes | IPDC | IP | IPv6 | IRC | ISAKMP | ISAKMP/IKE | iSCSI | ISTP | ISP | LDAP | L2F | L2TP | MARS | Mobile IP | MZAP | NARP | NetBIOS/IP | NHRP | NTP | OSPF | PIM | POP3 | PPTP | Radius | RLOGIN | RIP2 | RIPng for IPv6 | RSVP | RTSP | RUDP | SCTP | S-HTTP | SLP | SMTP | SNMP | SOCKS | TACACS+ | TALI | TCP | TELNET | TFTP | TLS | TRIP | UDP | Van Jacobson | VRRP | WCCP | X-Window | XOT

Additional Information

This Page is Inserted by IFW Indexing and Scanning Operations and is not part of the Official Record

BEST AVAILABLE IMAGES

Defective images within this document are accurate representations of the original documents submitted by the applicant.

Defects in the images include but are not limited to the items checked:

- □ BLACK BORDERS
- □ IMAGE CUT OFF AT TOP, BOTTOM OR SIDES
- □ FADED TEXT OR DRAWING
- □ BLURRED OR ILLEGIBLE TEXT OR DRAWING
- □ SKEWED/SLANTED IMAGES
- □ COLOR OR BLACK AND WHITE PHOTOGRAPHS
- □ GRAY SCALE DOCUMENTS
- □ LINES OR MARKS ON ORIGINAL DOCUMENT
- □ REFERENCE(S) OR EXHIBIT(S) SUBMITTED ARE POOR QUALITY
- □ OTHER: ______________________________

IMAGES ARE BEST AVAILABLE COPY.
As rescanning these documents will not correct the image problems checked, please do not report these problems to the IFW Image Problem Mailbox.